Bird Images Search Engine

Federica Baldi, Daniele Cioffo, Edoardo Fazzari, Mirco Ramo

MSc in Artificial Intelligence and Data Engineering

TABLE OF CONTENTS

Bird Images Search Engine

Federica Baldi, Daniele Cioffo, Edoardo Fazzari, Mirco Ramo
MSc in Artificial Intelligence and Data Engineering

1. INTRODUCTION

The purpose of this project is to create a search engine that can recognize the species of a bird from an image uploaded by the user, proposing also similar pictures. To this end, the following steps were performed.

First of all, our own version of the PP-Index was implemented in order to enable fast similarity search on deep features. In fact, both the images within the database and the user's queries are represented by feature vectors extracted by means of a Deep Neural Network, so that the similarity between them can be easily computed via a metric such as the Euclidean distance or the cosine similarity. As far as the Deep Neural Network is concerned, ResNet was used as the base architecture, first exploiting feature extraction, and then performing fine tuning of some layers.

Next, the retrieval performance of the image search engine was measured in both the feature extraction and fine-tuning cases, considering various metrics. Furthermore, a brief analysis was performed on the errors made by the system, including a focus on CNN's explainability. Finally, a web user interface[1] was built to allow the search engine to be used from a web browser.

1.1. Dataset

The dataset used to develop the search engine, *Birds-325*[2], contains images of birds belonging to 325 different species. It was prepared by Gerald Piosenka, a Kaggle user who gathered images from internet searches by species name, removed duplicates, and cropped the images so that each bird occupies at least 50% of the pixels. Next, the images were also resized to 224×224×3 and stored in .jpg format.

The dataset contains a total of 50582 images, which have already been split into training set, validation set and test set, as shown in Table 1.

Training Set	47332
Validation Set	1625
Test Set	1625
Total	50582

Table 1. Birds-325 dataset

It is important to note that both the validation and the test set contain 5 images for each of the species, while the training set is not perfectly balanced as it has a varying number of images per species. However, each of the species has at least 116 images in the training set.

Another important fact to highlight is the presence of a strong imbalance between images of female examples and images of male examples. In fact, only 15% of the images represent female birds, and most of the images in the validation

[1] The image search engine is available at https://birds-search-engine.anvil.app/
[2] https://www.kaggle.com/gpiosenka/325-bird-species

and test sets are of male examples. This poor representation of the female gender may lead to problems in its recognition because of the sexual dimorphism that, in birds, can be manifested both in size or plumage differences (ornamentation and/or coloration). For instance, Figure 1 shows two mallard duck examples. As can be seen, this species is characterized by a marked sexual dimorphism: males and females are very similar in shape but differ in plumage and beak color.

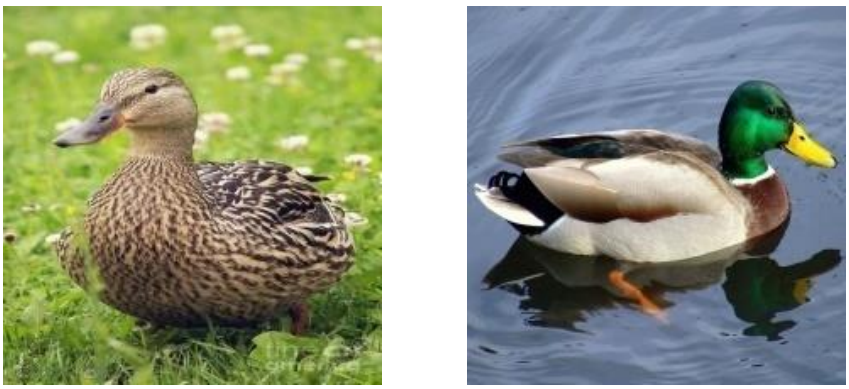


Figure 1. Female (left) and male (right) mallard ducks

In addition to the dataset containing bird images, the search engine database contains another dataset that serves the function of a "distractor": MIRFLICKR-25000[3]. It consists of 25000 images downloaded from the social photography site Flickr that represent different subjects, including landscapes, people, animals, and buildings.

As can be seen from Figure 2, these images are very different from what we want our search engine to be able to recognize. Therefore, what is expected is that if a user presents the search engine with an image that does not contain a bird, the system will retrieve images belonging to the distractor dataset.

Figure 2. Two images from the distractor dataset

[3] M. J. Huiskes, M. S. Lew (2008). The MIR Flickr Retrieval Evaluation. ACM International Conference on Multimedia Information Retrieval (MIR'08), Vancouver, Canada

2. PP-INDEX

The PP-Index (Permutation Prefix Index) is an index data structure that was first proposed by A. Esuli in "*PP-Index: Using Permutation Prefixes for Efficient and Scalable Approximate Similarity Search*" (2009)[4]. It belongs to the family of the permutation-based indexes, however it has a key difference with respect to previously presented PBIs: while such PBIs use permutations in order to estimate the real distance order of the indexed objects with respect to a query, the PP-Index uses the permutation prefixes in order to quickly retrieve a reasonably-sized set of candidate objects, which are likely to be at close distance to the query object, then leaving to the original distance function the selection of the best elements among the candidates.

Given a collection of objects D to be indexes, and the similarity measure d, a PP-Index is built by specifying a set of *pivots* P, and a *permutation prefix length* l. Any object $o_i \in D$ is represented by a *permutation prefix* w_{o_i} consisting of the first l elements of the permutation Π_{o_i}. The permutation Π_{o_i} consists of the list of identifiers of pivots, sorted by their distance with respect to the object o_i.

The prefix tree of the PP-Index is built on all the permutation prefixes generated for the indexed objects and is kept in main memory. The leaf at the end of a path relative to a permutation prefix w keeps the information required to retrieve the data blocks relative to the objects represented by w from the data storage.

A data block b_{o_i} contains both the information required to univocally identify the object o_i and the essential data used by the function d in order to compute the similarity between the object o_i and any other object. The data storage consists of a file in which all the data blocks are sequentially stored. The order of objects (represented by data blocks) in the data storage is the same as the one produced by performing an ordered visit of the prefix tree. This is a key property of the PP-Index, which allows to use the prefix tree to efficiently access the data storage.

2.1. Implementation

2.1.1. Data Structures

Our implementation of the PP-Index is based on two key data structures, namely the Node and the PrefixTree classes. The first represents the base building block of the tree, and it can refer both to the root and to a leaf and to an intermediate node. Either way, a Node stores:

- ***id***: pivot identifier corresponding to the current node within the prefix. It uniquely identifies the node within a path. For the root it is an empty string
- ***numeric path***: permutation prefix truncated at the current node. It uniquely identifies the node within the tree. For the root it is an empty string
- ***referenced objects***: number of objects referenced by the leaves that can be reached from the current node
- ***children***: list of children of the node. If the node is the root or an internal node, it is a list of nodes. In the case where the node is a leaf, it is a list of database object identifiers

A PrefixTree is a data structure composed by Nodes, it keeps the prefix tree and defines all the methods to access, traverse, modify, save permanently, and load from disk the index.

[4] Esuli, A. (2009b). Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. In *Proceedings of the 7th workshop on large- scale distributed systems for information retrieval. LSDS-IR '09* (pp. 17–24).

In this way the represented PP-Tree contains the information about the permutations of every object: each node represents, through its ***id*** a certain index in the permutation (the current level of the three specifies the position of the index in the permutation), the leaves, as previously explained, store also whatever is needed to identify the objects (o_i) and to compute the distances w.r.t to the query. A visual representation of our Prefix Tree is reported in Figure 3.

Green circle represents the root, which has no numeric ***id***; white squares are intermediate nodes and the number reported is their numeric ***id;*** light blue squares are leaf nodes, they store the ids of the indexed objects.

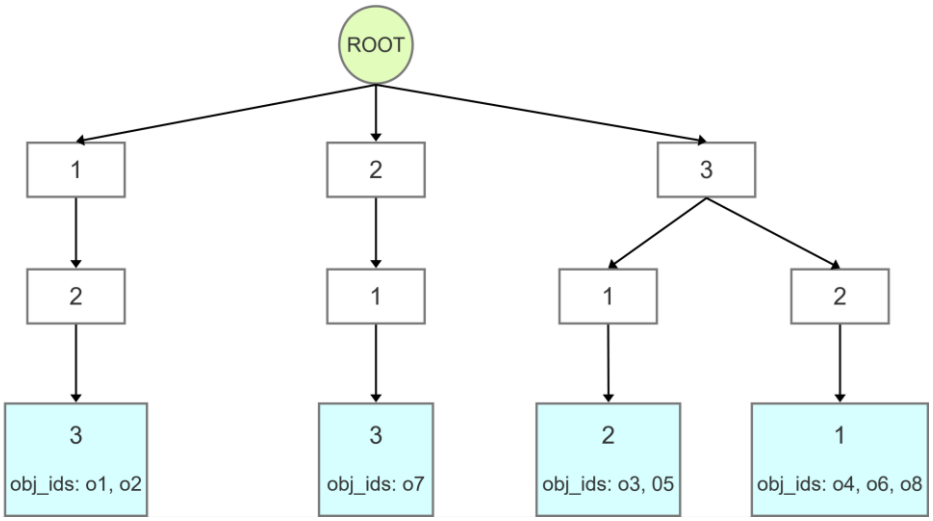


Figure 3. Prefix Tree structure

2.1.2.Implementation choices

With respect to the original paper, we introduced some changes in the physical tree implementation, concerning respectively the structure and the disk storage.

2.1.2.1 Tree structure

The original Prefix Tree contains one root that directly stores all the first level prefix values. Each of them also has a pointer to its children, which in turn is an array of second level prefix values and so on. The leaf stores three important fields, namely the ***referenced objects*** list and two $\boldsymbol{h_{start}}$ and $\boldsymbol{h_{end}}$ pointers to the portion of the disk where we are storing the descriptors of the objects with that prefix (b_{o_i}).

In our implementation, in order to create a simpler architecture, we decide to:

1. Standardize every node to a Node element, regardless of its position in the Tree
2. Allocate one node for each prefix number, and not multiple numbers in the same node, to allow a better management of the children
3. Compliantly with Python, we do not use pointers to children but only references
4. Leaves store only the ***referenced objects*** list, and have no need to use $\boldsymbol{h_{start}}$ and $\boldsymbol{h_{end}}$; indeed, the prefix is enough to reference the file in which object descriptors are stored
5. Every Node has a counter of referenced objects in the subtree, which is exploited at query time: whenever the algorithm must choose a node whose prefix is different from the query one, **it will choose the node that references the minimum number of objects** (we want to retrieve the **smallest** subtree that references at least k' objects)

2.1.2.2 Disk storage

In the original paper, a single file is used, storing objects with the same prefix in contiguous portions of the disk. We opted for a slightly different technique: we initialized one file for each permutation to store every object with that prefix in the file (e.g., if o_1 has prefix "123", its descriptor will be stored in "123.npy"). In this way we have that:

a. We are **not required to keep pointers in the leaves**, just the permutation is enough to find where to seek on the disk
b. Tree building is for sure **faster**, since we don't need to seek the position where to write in the file, but we apply a simple append. Writing many times in the middle of a file could cause many disk relocations, slowing down considerably the I/O operation
c. From the point of view of code simplicity, this approach is much more readable and **less error-prone**, and allows for a better organization of data
d. If objects with the same prefix of the query are enough (≥k′), K-NN candidates retrieval is **faster** because again we don't need to seek the file position and read from a very big file, but just read the whole content of a medium-sized file
e. If objects with the same prefix of the query are not enough (<k′), K-NN candidates retrieval is **slower** because this time we will have to access multiple files instead of reading from contiguous portions of the disk

2.1.3. Building the Index

Tree building is a very expensive operation since it requires to compute the distances between objects and pivots for every single object in the database. Anyway, this procedure is performed only once: the tree is then stored in the disk and loaded in the main memory at launch time. Starting from an object list, we call recursively the *insert* method, which is in charge to compute the prefix of each object and indexing it in the three, if necessary, creating needed nodes and leaves.

2.1.4. Search Function

The search function is performed at query time, whenever the user must satisfy an information need. The most important properties of a search function are reduced ***query time*** and ***relevance*** of the results. In our application, it is meaningless to implement exact searches, so the only paradigm allowed is the ***search-by-example.*** We decided to implement only the k-NN query, thus ignoring the range query, because the latter is not suitable to be performed with a PP-Index and it is not so used in practice. To query the database exploiting our index, we just need to call the *find_nearest_neighbors* procedure, which is in charge of computing the query prefix, retrieve all the candidates, reorder them, and give as output the *k* nearest neighbors. In order to address and optimize the search function, we converged to the following decisions:

- *Optimize relevance*: the policy explained at 5 make the algorithm retrieve a set of objects composed by the joining of a higher number of permutations, thus increasing the probability to find objects close to the query but with a slightly different permutation. In this phase, sort the children list by increasing number of referenced objects, so that simply scanning the children list automatically means visiting them starting from the one with the lowest number of referenced objects
- *Optimize query time*: we decided to statically fix the parameter c (amplification factor) to 5 (# of retrieved candidates $k' = 5 \cdot k$), which is quite a low value. Lower k' means lower number of retrieved candidates and

thus more efficiency at the possible expenses of the effectiveness. We also fixed the length of the prefixes (l) to 3, again a low value, reducing the height of the tree and then allowing for faster traversals

2.2. Improvements

As pointed out in the original paper, the "basic" PP-Index is strongly skewed towards efficiency, placing effectiveness on the back burner. The two measures, as usual, are conflicting and finding the right compromise between the two is crucial. In order to improve the effectiveness of the PP-Index we have implemented the two "boosting" strategies: *multiple indexes* and *multiple queries*.

Regarding the first strategy, t indexes are built based on different $P_1 \dots P_t$ sets of pivots. Hence, we do not have a single *prefix tree*, but we have a *"forest"* containing t trees. When a query is submitted, the set of candidate objects is the union of the sets of candidate objects obtained using all t indexes. The strength of this strategy is that it somehow bypasses the problem of choosing pivots, as a single choice might be suitable for one query and not another. The disadvantage is that we need to store objects t times, one per *prefix tree*.

The second strategy, on the other hand, requires that, at search time, n additional permutation prefixes are generated from the permutation prefix of the query. In particular, new prefixes are generated by adding perturbation to the original one, i.e., swapping the positions of some of its elements. Again, when a query is performed, the set of candidate objects is the union of the sets of candidate objects obtained for each of the $n + 1$ queries.

The two strategies were also combined, i.e., each time a query is submitted, first the n additional permutation prefixes are generated and then all queries are executed on each of the t trees that are part of the *prefix forest*.

In an effort to maintain good efficiency and reduce computation time, multithreading was used when applicable in order to parallelize the search process as much as possible.

3. RESNET

ResNet is one of the most powerful deep neural networks which has achieved excellent results in the ILSVRC 2015 classification challenge. ResNet has achieved excellent generalization performance on other recognition tasks and won the first place on the ImageNet detection, ImageNet location, COCO detection and COCO segmentation in ILSVRC and COCO 2015 competitions. There are many variants of ResNet architecture, each one with a different number of layers. It was created with the aim of tackling the problem of degradation in deep neural networks, with the use of residual blocks. In this work we will use ResNet-50v2, ResNet-101v2 and ResNet-152v2.

3.1. Feature Extraction

First, we used the pretrained architectures to extract the features of our images, from the *GlobalAveragePooling2d* layer of the ResNet-50v2, ResNet-101v2 and ResNet-152v2 architecture. As we will see later, the performance with these features is not very good. In fact, our architectures are pretrained on the *ImageNet* dataset, where all birds end up in very few classes, and thus with very similar features to each other. Next, we moved on to develop MLPs on top of the extracted features with the pretrained convolutional layers.

3.1.1. ResNet-152v2

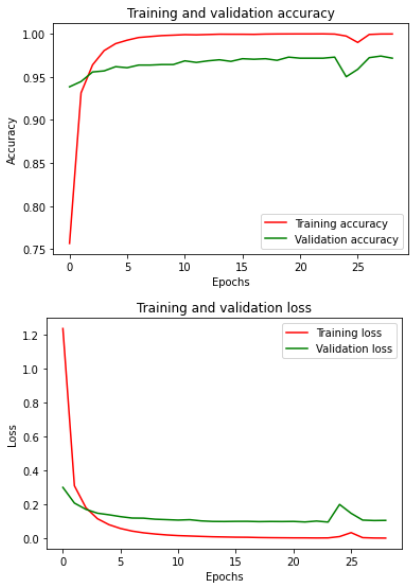
We will now report the best classifiers obtained on top of the ResNet-152v2 pretrained features.

3.1.1.1 Experiment 1

In this experiment we just decided to add the classification final dense layer of 325 neurons, with SoftMax activation function. Afterwards, we performed the training of this last layer.

Training loss	Training accuracy	Validation loss	Validation accuracy
0.0025	0.9997	0.0957	0.9729

Table 2. Training and validation results




```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet152v2 (Functional)     (None, 7, 7, 2048)        58331648

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

predictions (Dense)          (None, 325)               665925

=================================================================
Total params: 58,997,573
Trainable params: 665,925
Non-trainable params: 58,331,648
```

Figure 4. Learning curves and summary of the network

Even with this very simple architecture, we achieved some incredible results. In fact, the features extracted with *ImageNet* weights concern the characteristic properties of our bird species, resulting in an excellent classification. You can notice some overfitting, but very small.

Accuracy	Weighted F1-score	Test loss
0.9754	0.9749	0.0795

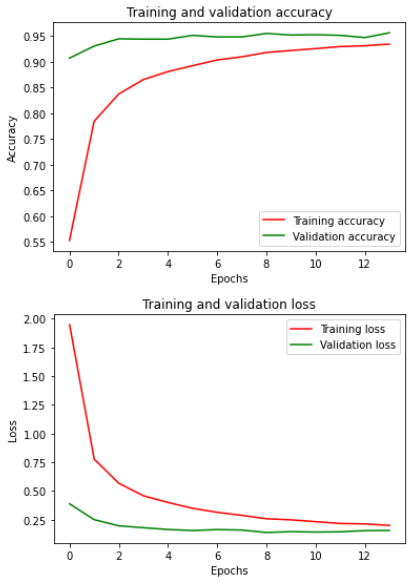
Table 3. Results on the test set

3.1.1.2 Experiment 2

We tried to insert another dense layer before the classification one, with 512 neurons and ReLU as activation function. To fight the overfitting, we also added Dropout as regularization technique. The main idea is to try to decrease the feature size as far as possible, seeing if we can still get good performance. In fact, it is important to remember that we are not looking for the best classification model, but the best model for our retrieval system. A good classification model will certainly help in this process, but feature size is a key element of our analysis.

Training loss	Training accuracy	Validation loss	Validation accuracy
0.2589	0.9181	0.1393	0.9551

Table 4. Training and validation results




```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet152v2 (Functional)     (None, 7, 7, 2048)        58331648

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 512)               1049088

dropout (Dropout)            (None, 512)               0

predictions (Dense)          (None, 325)               166725

=================================================================
Total params: 59,547,461
Trainable params: 1,215,813
Non-trainable params: 58,331,648
_________________________________________________________________
```

Figure 5. Learning curves and summary of the network

The results are still great, but with features four times smaller. Note that the validation curve is better than the training curve, this is due to the use of dropout, which acts only during training delaying the curve.

Accuracy	Weighted F1-score	Test loss
0.9631	0.9626	0.1172

Table 5. Results on the test set

3.2. Fine-Tuning

Fine tuning is the operation of unfreezing layers from the top of the network to make them trainable, in the way their weights are changed due to backpropagation. This section describes the results obtained by fine tuning starting from different layers in our networks.

3.2.1.ResNet-50v2

ResNet-50v2, due to its low depth compared to the other two architecture, has a very fast training even if the fine tuning is protracted to consider a huge number of convolutional layers, thus increasing the number of trainable parameters. Therefore, for this architecture it was possible to fine-tune multiple times in the search of the best in terms of accuracy on the validation set. The results of the tests are reported in Table 6:

Fine-tuned from	Validation Accuracy	Test Accuracy	Validation Loss	Test Loss
Conv5_block2_out	0.7489	0.7612	1.3761	1.1639
Conv5_block1_out	0.7594	0.7840	1.2738	1.0885
Conv4_block6_out	0.7618	0.7840	0.9477	0.7805
Conv4_block5_out	0.8240	0.8388	0.8948	0.7162
Conv4_block4_out	0.8252	0.8332	0.8647	0.6962
Conv4_block3_out	0.8351	0.8720	0.7992	0.5360
Conv4_block2_out	0.8274	0.8720	0.8015	0.5360

Table 6. Fine Tuning Results for ResNet50v2

3.2.2.ResNet-101v2

Due to the size of ResNet-101v2 we only attempt fine-tuning starting from two different blocks located in *conv4*. The choice of these two blocks was driven by the previous experiment and the results obtained are reported in Table 7.

Fine-tuned from	Validation Accuracy	Test Accuracy	Validation Loss	Test Loss
Conv4_block19_out	0.7858	0.8142	0.9016	0.7106
Conv4_block13_out	0.8265	0.8615	0.7535	0.8615

Table 7. Fine-tuning Result for ResNet101v2

3.2.3.ResNet-152v2

Finally, we analyze the results obtained by performing fine tuning on our best models of the ResNet-152v2. This architecture is very large, and in fact we were often only able to unfreeze a few levels due to Colab's limitations.

3.2.3.1 Experiment 1

As previously mentioned, with this model we have only added the last level of classification with 325 neurons, and the features will be extracted directly from the output of the global average pooling layer. Through fine tuning we will now modify the weights of the convolutional layers, adapting them to our dataset. With this model we tried to unfreeze both the fifth block levels and also the fourth block levels.

Fine-tuned from	Training loss	Training accuracy	Validation loss	Validation accuracy
Conv5_block1_1_conv	$9.7086e^{-4}$	0.9999	0.1023	0.9692
Conv4_block1_1_conv	$5.0013e^{-4}$	1.0000	0.0993	0.9742

Table 8. Training and validation results

In Figure 6 both results are shown, on the left unfreezing the fifth block and on the right unfreezing also the fourth block.

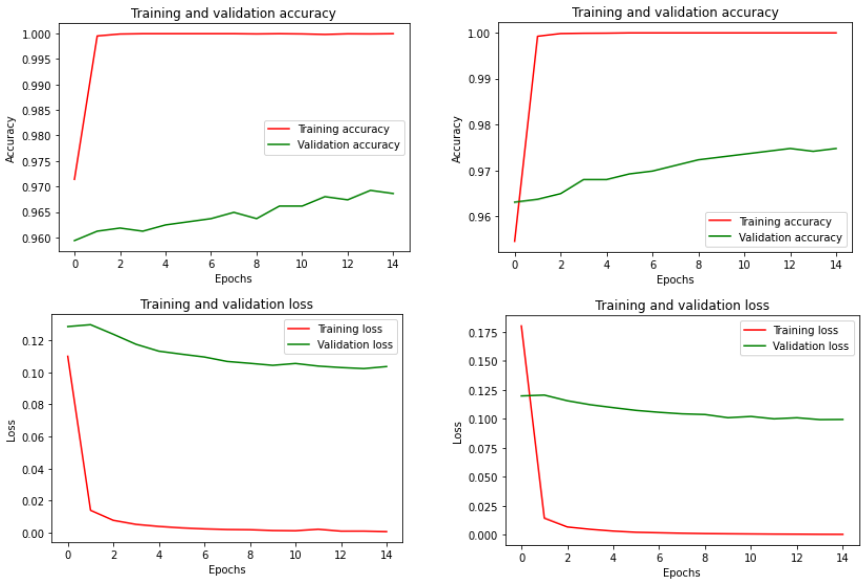


Figure 6. Learning curves unfreezing 5-th block and 4-th block

The overfitting is very small even in the first case, but thanks to more fine tuning we were able to fight it even more.

Fine-tuned from	Accuracy	Weighted F1-score	Test loss
Conv5_block1_1_conv	0.9760	0.9756	0.0831
Conv4_block1_1_conv	0.9828	0.9826	0.0703

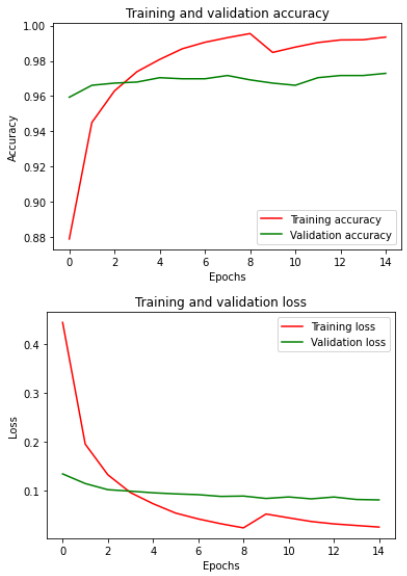
Table 9. Results on the test set

3.2.3.2 Experiment 2

As previously mentioned, with this model we added another dense layer of 512 neurons before the classification layer, using dropout as regularization technique. We could not unfreeze also the fourth block because of some limitations of Colab, we were exceeding the amount of RAM of the GPU assigned to us and to solve the problem we had to decrease the batch size, resulting in increased training time. This solution was not feasible anyway, because it led us to exceed the 8 hours of execution allowed.

Fine-tuned from	Training loss	Training accuracy	Validation loss	Validation accuracy
Conv5_block1_1_conv	0.0258	0.9935	0.0813	0.9729

Table 10. Training and validation results




```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0

tf.math.truediv (TFOpLambda  (None, 224, 224, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 224, 224, 3)       0
a)

resnet152v2 (Functional)     (None, 7, 7, 2048)        58331648

global_average_pooling2d (G  (None, 2048)              0
lobalAveragePooling2D)

dense (Dense)                (None, 512)               1049088

dropout (Dropout)            (None, 512)               0

predictions (Dense)          (None, 325)               166725

=================================================================
Total params: 59,547,461
Trainable params: 16,184,645
Non-trainable params: 43,362,816
```

Figure 7. Learning curves and summary of the network

Thanks to dropout the overfitting is very limited, and the validation curve follows very well the training curve.

Fine-tuned from	Accuracy	Weighted F1-score	Test loss
Conv5_block1_1_conv	0.9822	0.9820	0.0673

Table 11. Results on the test set

The results obtained are amazing, we reduced the dimensionality of the features by a factor of 4 and still the network is able to achieve almost identical results on the test set.

4. PERFORMANCE EVALUATION

In this chapter we will measure the retrieval performance of the image search engine created on top of our PP-Index implementation, using both features obtained from the pretrained DNN and those obtained as a result of fine tuning. Each time, the performance obtained with the index will be compared with the one obtained by applying the exhaustive computation of object-to-query distances before ranking (brute force method).

In addition, the performance evaluation will allow us to incrementally obtain the best configuration and make the choices of the hyper-parameters such as:

- The model of ResNet (both pretrained and finetuned) to be used for extracting features
- The type of distance (or similarity) metric to be used on the extracted features
- The criterion for selecting the pivots
- The number of pivots to be selected
- The optimization(s) to be used (*multiple index* and/or *multiple query*)

Eventually, we analyzed further possible improvements exploiting Autoencoders.

In order to measure performance, we selected 325 images (one per species) from the test set of the *Birds-325* dataset and we used two performance indicators: the *average query time* and the *mean Average Precision* (*mAP*). The *average query time* is simply the average time taken to return the results of each of the 325 nearest neighbors queries, and the *mAP* formula is as follows:

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$

where N is the number of queries that were made and AP is the *Average Precision*. The latter was calculated as follows:

$$AP@n = \frac{1}{GTP}\sum_{k=1}^{n} P@k \times rel@k$$

where GTP refers to the total number of ground truth positives (i.e., the total number of images of the same species as the query within the database), n refers to the total number of images we are interested in, $P@k$ refers to the $precision@k$ and $rel@k$ is a relevance function. The relevance function is an indicator function which equals 1 if the image at rank k is relevant and equals to 0 otherwise. As for n, we want it to be larger than the number of images per species, so that the AP can reach 1 in case all relevant images are retrieved first. Therefore, since at most we have 249 images per species, we chose $n = 250$.

4.1. Feature Extraction

First, we are going to analyze which is the best feature extraction model, starting the analysis without the distractor. After, with the addition of the distractor, we will insert some noise into our system, to test if the extracted features are indeed robust.

4.1.1. Without Distractor

4.1.1.1 ResNet pretrained model for extracting features

In the previous analysis, we saw three different architectures of ResNet, each of which extract different features from the images. In fact, some of them are deeper, while others stop at lower-level features. This is why we had to measure the

performance of all architectures, analyzing which one is the best for our retrieval system. To decide the best model, we compared the performance with brute force, trying to understand which architecture was the most promising.

As we can easily see in Table 12, the best feature extraction model is ResNet50v2, but in general the results are very poor.

Model	Distance/similarity metric	Mean Average Precision	Average query time (s)
ResNet50v2	Cosine similarity	0.00694	19.2
ResNet101v2	Cosine similarity	0.00481	13.8
ResNet152v2	Cosine similarity	0.00565	13.5

Table 12. Comparison between ResNet models

4.1.1.2 Distance or similarity metric

Since the features extracted by means of the ResNet50v2 model resulted in better performance, we decided to build the PP-Index on them. Next, we evaluated which metric was best to use: Euclidean distance or cosine similarity. To this end, two indices were constructed with 5 randomly chosen pivots. Please note that at this point we do not know if this is the best configuration, we just need it to compare the two metrics fairly.

Distance/similarity metric	Mean Average Precision	Average query time (s)
Cosine similarity	0.00499	0.750
Euclidean distance	0.00388	2.129

Table 13. Comparison between Euclidean distance and cosine similarity

As can be seen from Table 13, cosine similarity leads to a higher *mAP*. This result is presumably due to the fact that the 2048-component descriptors were not normalized and thus the Euclidean distance may be high for descriptors that are actually similar (i.e., have a similar distribution of component values) but have different length. On the other hand, cosine similarity overcomes this problem by performing normalization of the vectors.

With this basic configuration we see how PP-Index has on the one hand an *Improvement in Efficiency* (IE) of about 25 (considering average query time as a cost) but, on the other hand, a worsening of the *mAP* of about 30% compared to the brute force approach. Future experiments will help us to better tune the tradeoff between efficiency and effectiveness.

4.1.1.3 Pivot initialization strategy

A crucial factor over the index performance is the pivot choice: pivots affect the way the Prefix tree is built and traversed, so they have a strong impact especially on the effectiveness. In order to decide which strategy suits best our implementation, we tested two different strategies (namely Random and K-Medoid), choosing the best one based on ***mAP, query time*** and only secondly ***building time***. Results are reported in the Table 14, for this test we exploited a simple Prefix Tree with 5 pivots and the cosine similarity.

Pivot Strategy	Mean Average Precision	Average Query time [s]	Building time (index + pivots) [s]
Random	0.0050	0.750	57 + 0
K-Medoid	0.0049	1.269	48 + 201

Table 14. Performance with different pivot strategies

Important observation: for the random strategy, we performed more than one test, but we only reported the worst achieved values (worst scenario case). On the contrary, K-Medoid performance kept roughly constant among the tests.

We immediately notice that effectiveness results are practically the same, just slightly better the Random strategy, but efficiency measures are not: the tree randomly initialized is much faster both at query and at building time. We believe that those experimental results have been affected by the frequent oscillations in the performance of the resources offered by Google Colaboratory, however the fact that K-medoid tends to prefer more balanced trees, with similar number of refenced objects in each leaf, is reflected in the times: Random is faster at query time because it accesses a lower number of files to retrieve candidates; it is slower in inserting the objects in the tree but the total building time is still much lower because of the big amount of time needed by K-medoid to find centroids. In addition, K-medoid introduces a scalability issue: an instance of the algorithm performed over 47K elements consumed 11.9GB of RAM, making it impossible to run over larger datasets (e.g., the dataset with the distractor).

Given all these considerations, we deem the Random Strategy as the best one, and all the following evaluations will be performed with this kind of pivot initialization.

4.1.2.With Distractor

In order to evaluate the robustness of the features extracted from the pretrained ResNet50v2 model, we repeated the brute force experiment using the cosine similarity with the addition of the distractor. As we expected, and as shown in Table 15, results are worse than the previous ones. However, since the worsening is small, we can consider the features "robust" in that they manage to differentiate the birds from the noise (despite the poor performance). Please note that from here on within the tables we show within brackets the improvement or deterioration from the previously tested best configuration.

Model	**Distance/similarity metric**	**Mean Average Precision**	**Average query time (s)**
ResNet50v2	Cosine similarity	0.00596 (− 0.00098)	21.5 (+ 2.3)

Table 15. Brute force with Distractor

The test to assess which metric was better was also repeated with the addition of the distractor. This confirmed that the cosine similarity performs better for our task. Moreover, even for PP-Index we can see a small worsening in the *mAP* and average query time, despite the addition of 25 thousand more images. In Table 16, we report the results over the Prefix Tree initialized with 5 Random pivots, showing the *mAP* and the *Average Query Time* when considering Euclidian distance or Cosine similarity as metric.

Distance/similarity metric	**Mean Average Precision**	**Average query time (s)**
Cosine similarity	0.00435 (− 0.00064)	1.904 (+ 1.154)
Euclidean distance	0.00296 (− 0.00092)	1.677 (− 0.452)

Table 16. Comparison between Euclidean distance and cosine similarity with Distractor

4.1.3.MLP

The results with feature extraction alone are poor. As previously mentioned, in the *ImageNet* dataset the birds all end up in few classes; in fact, these pretrained architectures are meant to understand if the animal is a bird, but not the

difference between species. This results in poorly distinguished features that lead to poor results. Now we are going to test the results obtained by adding the dense layer of 512 neurons on top of the convolutional base of the ResNet-152v2, which is the architecture that performs best with the addition of an MLP.

We can easily see that the features are robust, with little difference in performance between the case with distractor and the case without. Again, we continue to see the improvements that cosine similarity brings over Euclidean distance. But the most incredible results are those involving *Mean Average Precision*. These results are now satisfactory, and we expect further improvement with fine tuning.

Distance/similarity metric	**Distractor**	**Mean Average Precision**	**Average query time (s)**
Cosine similarity	NO	0.60430 (+ 0.59865)	13.142 (− 0.358)
Cosine similarity	YES	0.60103 (+ 0.59507)	20.039 (− 1.461)
Euclidean distance	NO	0.54535	10.102
Euclidean distance	YES	0.54167	15.172

Table 17. MLP results with brute force

At this point we tested the performance of the index built on the features extracted from the dense layer. The test was performed by going for the best configuration seen so far, i.e., 5 randomly chosen pivots and cosine similarity as a metric. As Table 18 shows, there was in an improvement in both *mAP* and *average query time*. The latter is likely due to the reduction in the dimensionality of the descriptors from 2048 to 512.

Distance/similarity metric	**Mean Average Precision**	**Average query time (s)**
Cosine similarity	0.17707 (+ 0.17272)	0.645 (− 1.032)

Table 18. PP-Index results on the features extracted from the MLP

4.2. Fine Tuning

In this chapter the performances are described using the descriptors obtained by extracting the features from the *GlobalAveragePooling2D* of the fine-tuned ResNet152v2 and those from the 512-units dense layer of the classifier built on top of the network.

4.2.1.Without Distractor

First of all, we performed tests using the brute force approach without the distractor in order to have a basis of comparison to evaluate the robustness of the extracted features. Tests were performed on both the features extracted from the *GlobalAveragePooling2D* of the fine-tuned ResNet152v2 and those from the 512-units dense layer of the classifier built on top of the network. In addition, both metrics were tested: Euclidean distance and cosine similarity.

Extracted from	**Distance/similarity metric**	**Mean Average Precision**	**Average query time (s)**
GlobalAveragePooling2D	Euclidean distance	0.37077	10.4
GlobalAveragePooling2D	Cosine similarity	0.49835 (+ 0.49)	14.8
512-units dense layer	Euclidean distance	0.69539 (+ 0.15)	10.1
512-units dense layer	Cosine similarity	0.76834 (+ 0.16)	14.4

Table 19. Performance of the brute force approach over features extracted by different fine-tuned models without distractor

4.2.2.With Distractor

4.2.2.1 Layer from which to extract features and distance or similarity metrics

Tests previously performed without the distractor were repeated with the purpose of evaluating the robustness of the features, and also deciding from which layer to extract them and which metric to use.

As we can see from Table 20, the *mAP* worsened, however only slightly and this confirms the robustness of our features. The results of these tests also showed that the best features are obtained using the 512-units dense layer along with the cosine similarity. For this very reason, subsequent analyses will use this model and this metric.

Extracted from	**Distance/similarity metric**	**Mean Average Precision**	**Average query time (s)**
GlobalAveragePooling2D	Euclidean distance	0.36997 (− 0.00080)	13.9 (+ 3.5)
GlobalAveragePooling2D	Cosine similarity	0.49673 (− 0.00162)	21.6 (+ 6.8)
512-units dense layer	Euclidean distance	0.68658 (− 0.00881)	17.3 (+ 7.2)
512-units dense layer	Cosine similarity	0.76446 (− 0.00388)	19.9 (+ 5.5)

Table 20. Performance of the brute force approach over features extracted by different fine-tuned models with distractor

4.2.2.2 PP-Index improvements

In the previous evaluation, we found out which model is the one that allows for highest performance, namely the ResNet152v2 with 512 neurons in the dense layer. Now we take the descriptors generated by it and we use them to build a PP-Index, comparing then the performance in terms of efficiency boosting but also effectiveness degradation.

In this experiment we do not only take into consideration the standard index, but we also consider the improvements introduced at chapter 2.2, trying to evaluate which index structure is the most performing one: for all of them we carry out the test considering only the cosine similarity (yet proved to be the best one in previous experiments), and taking for all of them the same set of 5 random pivots (5x3 for forests). Forests have been built as collections of 3 trees, query perturbations are again 3 per query.

For each index, we report the *Mean Average Precision* and the *Average Query Time*, but also the ratio of mAP with respect to the brute force ($mAP(index)/mAP(brute\ force)$), and the relative Improvement in Efficiency

$IE = Avg\ Query\ Time(brute\ force)/Avg\ Query\ Time(index)$.

Improvement	**mAP**	**mAP ratio with brute force**	**Average Query time (s)**	**IE**
No improvement	0.15720	20.56%	0.528	37.69
Query perturbations	0.25852	37.31%	1.297	15.34
Forest	0.40501	52.98%	2.077	9.58
Forest with query perturbations	0.52814	69.09%	4.297	4.63

Table 21. Performance comparison for different index improvements

As we immediately notice, the benefits of the indexes in the retrieval time are considerable, for the simple tree with no improvement, it takes almost 40 times less to perform a query; of course, this ratio decreases if we introduce further mechanisms in the index that require a higher number of distance computations, but even with multiple trees and multiple query perturbations, the speed-up factor is over 4.

As a trade-off, faster indexes tend to be imprecise: the simple tree without improvements has only one fifth of the *Mean Average Precision* with respect to the brute force approach.

Given those considerations, we decided to prioritize the effectiveness, thus considering the Forest with Query Perturbations as our best index, which reached a pretty good result in the *mAP*, but still allowing for a high value of *Improvement in Efficiency*.

4.2.2.3 Number of pivots

Thanks to the previous tests, we were able to state that the combination of the two improvements "forest" and "query perturbation" yields the best results in terms of *mAP*. The last hyperparameter we had left to optimize was the number of pivots to be used for the index construction.

We therefore chose to test the performance of the index using 5, 10, 20 and 40 pivots. Regarding the length of the permutation prefix, l, after some preliminary tests we chose to fix it at 3. In fact, this parameter does not seem to affect the performance of the index and setting it at a low value allows for a shallower tree that is therefore more quickly traversable. We decided not to test higher numbers of pivots because we wanted to avoid having a number of possible permutation prefixes higher than the number of images within our database. The number of l-permutations of n is given by the following formula:

$$P_{n,l} = \frac{n!}{(n-l)!}$$

For $n = 40$ pivots and $l = 3$ as the prefix length, we have $P_{40,3} = 59280$ partial permutations. In this way, even in the worst case the number of objects having certain permutation prefix is ≥ 1.

As we expected, and as Table 22 shows, the results improve as the number of pivots increases. Furthermore, with 40 pivots we are able to achieve a *mAP* that is only 11% worse than that achieved with brute force, while reducing the *average query time* by more than 5 times. Please note that these times are for a 250-nearest-neighbors search, typically we expect the search engine user to choose a smaller k value.

Number of pivots	Mean Average Precision	Average Query time (s)
5	0.52814	4.297
10	0.56829	3.460
20	0.60712	3.168
40	0.67832	3.553

Table 22. Performance comparison for different numbers of pivots

4.3. Autoencoder

For reducing the dimensionality from 512 to a lower dimension an autoencoder was introduced. We implemented two types of autoencoder: one which reduces the features from 512 to 256 and the other one from 512 to 128. The structure of the two networks is reported in Figure 8 and Figure 9.

The two architectures were trained for two different problems: for the features obtained from the finetuned ResNet152v2 with the 512 dense layer and the MLP classifier used in feature extraction.

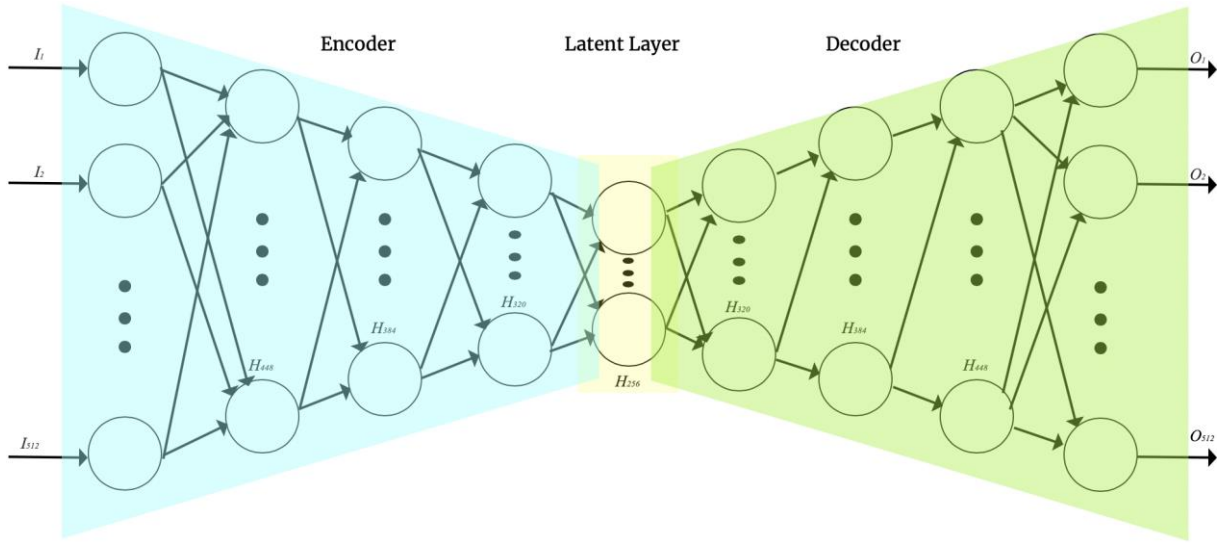


Figure 8. Autoencoder from 512 to 256

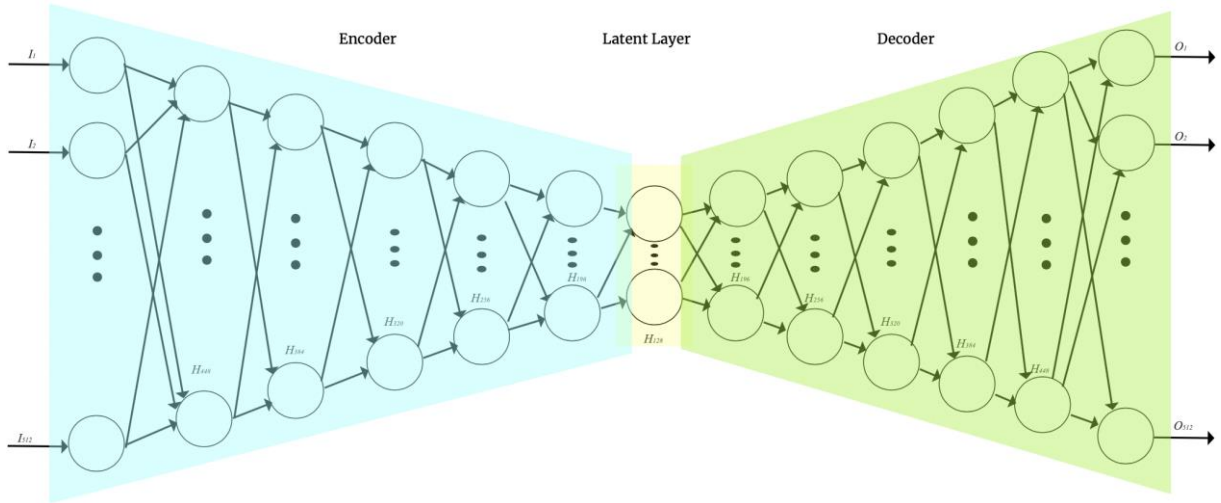


Figure 9. Autoencoder from 512 to 128

In order to train this type of network, we had to decide which loss and optimizer function to use. As it regards the loss we have chosen to use the mean square error, as the literature suggests, which is computed as follows:

$$\mathrm{MSE}(y,\ \hat{y}) = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2$$

where y_i is an input vector, and $\hat{y}_i$ is the related predicted value.

For choosing the optimizer, we opted to use the one proposed by the literature, thus the stochastic gradient descent (SGD).

4.3.1.Feature Extraction (MLP)

The first task to be accomplish was to train the two architectures using the features extracted by the MLP. In this operation, the features obtained from the training set were used as the new training set, likewise for the features obtained from the validation and testing sets.

The trained networks present the results shown in Table 23. The losses obtained were low for both the architectures, indicating that a good reconstruction of the features was obtained. Thus, both the features obtained by the two autoencoder can be used for the index.

Dimension	Training Loss	Validation Loss
256	0.4414	0.4174
128	0.6407	0.6122

Table 23. Autoencoder losses for feature extracted features

Since in the previous tests presented the cosine distance was the one performing better, the test here are done using only this distance. The first experiment we did was to use the brute force method in order to evaluate how the two reduced features performed. The results of this brute force analysis are reported in Table 24.

The results obtained with the 128-dimension features are lower than those with 256 but not by too much, so counting the memory saving we chose to use the 128-dimension version in the next tests.

Dimension	Mean Average Precision	Average Query Time
256	0.59522	22.0444
128	0.55997	20.9843

Table 24. Autoencoder MLP result for brute force

Since the 128-dimensional features succeed in achieving good results we decided to build our index on them. In fact, a smaller size can be beneficial both for the calculation of distances and for reducing the space occupied by the index itself. Table 25 shows the results obtained with the PP-Index built on these features using the best configuration, i.e., with 40 randomly chosen pivots, cosine similarity as a metric, and the use of both optimizations (*multiple index* and *multiple query*). We also reported the ratio of *mAP* with respect to the brute force and the *Improvement in Efficiency*.

mAP	mAP ratio with brute force	Average Query time (s)	IE
0.51848	92.59%	3.4215	6.13

Table 25. PP-Index built on features extracted from the Autoencoder MLP results

4.3.2.Features from the 512-units dense layer of the Finetuned ResNet152v2

Also in this case, the first task to be accomplish was to train the two architectures using the features extracted by the 512-units dense layer. In this operation, the features obtained from the training set were used as the new training set, likewise for the features obtained from the validation and testing sets.

The trained networks present the results shown in Table 26. The losses obtained were low for both the architecture, indicating that a good reconstruction of the features was obtained.

Dimension	Training Loss	Validation Loss
256	0.4460	0.4181
128	0.5428	0.5109

Table 26. Autoencoder losses for fine-tuned features

Also in this case, we reduced the features of the MLP to 256 and 128 and tested them using brute force. The results are reported in Table 27.

Dimension	Mean Average Precision	Average Query Time (s)
256	0.76378	21.25843
128	0.75037	19.95545

Table 27. Autoencoder finetuned results for brute force

The 128-features have a *mAP* value very similar to the one obtained from the 256-features ones, moreover the *average query time* is lower, indicating not only a reduction in memory used to save the features but also a lower computational cost. For this reason, the version with 128-features was used from now on regarding the finetuned architecture.

Once again, we tested the performance of our index with the best configuration and reported the results in Table 28. These are also the best results ever achieved with the index.

mAP	mAP ratio with brute force	Average Query time (s)	IE
0.67741	90.28%	3.4312	5.82

Table 28. PP-Index built on features extracted from the finetuned Autoencoder results

4.4. Analysis of the Best Configuration

In the chapter 4.2.2.3 we proved the Prefix Forest with query perturbation, cosine similarity and 40 random pivots to be our best index. Jointly, the chapter 4.3.2 showed that the fine-tuned ResNet152v2 with Dense layer and Autoencoder is the most accurate feature extractor, and it is also capable of shrinking the descriptors' size to allow for faster distance computations. Thus, we can conclude that the combination of those 2 elements are the building blocks of our best configuration. In this chapter we are going to recap all the performance of the index over the extracted features, in terms of Mean Average Precision, Average Query Time, Setup Time, but also Number of Required Distance Computations, Precision@k and Recall@k for some possible values of k that are typically requested by users and provided by our final application.

4.4.1. Mean Average Precision and Average Query Time

Table 28 already showed Mean Average Precision and Average Query Time of our configuration, and we already discussed, they are very good since we managed to reach more than 90% of the mAP that we achieve computing exhaustively all the query-object distances, but almost 6 times more quickly.

For what concerns the Average Query Time moreover, we computed it considering $k = 250$, that is retrieving at least 750 candidates: in the real application users can select 50 as maximum value of k, and seldom they wonder to use big values. Repeating the experiment taking smaller and more reasonable k values (5 to 40), we observe that the Average Query Time taken by the index is very **often below 1 second**.

Actually, for the purposes of the final application we have to consider also the time taken by the Deep Learning model to extract the features from an input image: from some experimental results we have that this time is usually around 0.5 seconds, so is common for the application to give back the result to the user in more or less **2 seconds.**

4.4.2.Setup Time

Setup Time is composed by two factors: index building and model loading.

	Model Loading	Forest Building	Total
Index Not Stored	25.14	220.81	245.95
Index Stored		0.59	25.63

Table 29. Setup time for the best configuration. All the measures are expressed in seconds

As we can see from Table 29, the use of the index could potentially require a lot of time before allowing to make the server side of the application operative: it is practically not true because, if it is retrieved from disk, the whole setup time takes only 25 seconds.

4.4.3.Number of Required Distance Computations

An important factor of the index under analysis is the number of distance computations that it requires at query time. This measure has not been computed experimentally, but we now estimate its amount, and we compare them with the one of the brute force approach.

The brute force approach requires to explicitly calculate all the *query-object* distances: for 1 query and 72332 database elements we have 72332 distance computations, regardless the value of k.

On the contrary if we use the index, we have just to consider the number of retrieved candidates: the number of query-object distance computations corresponds to this value. About this fact, we should consider that:

- The search stops when the algorithm finds at least k' candidates, it means that it could potentially retrieve more objects than k', but for simplicity we don't consider that
- When exploiting the improvements, there is the possibility to have duplicate candidates, in such case the algorithm merges them and computes the distance just once; again, we are not going to take this into account

Figure 10 shows the amount of distance computations needed with and without the use of the index, in a **logarithmic scale**: as we can see exploiting our forest with perturbations the amount of considered distances is hundreds of times lower than the brute force approach, even for very large k values.

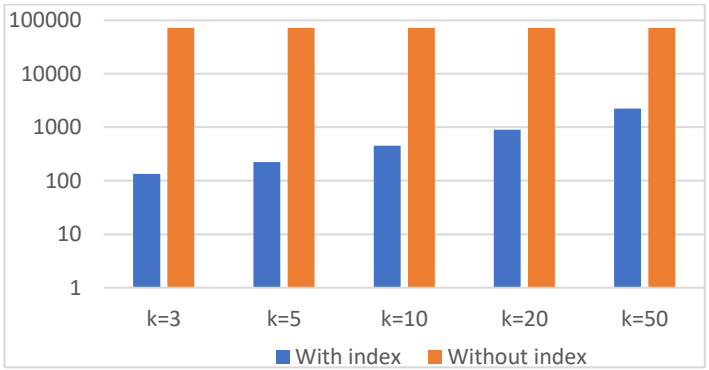


Figure 10. Amount of distance computations, logarithmic scale

4.4.4.Precision@k and Recall@k

Precision@k and Recall@k are two measures used to evaluate the performance of an approximate nearest neighbor search, they are defined respectively as:

$$P@k = \frac{Relevant\ Retrieved\ Documents}{k} \qquad R@k = \frac{Retrieved\ Documents \cap Groundtruth\ NN}{k}$$

Those two measures are very useful to determine the performance of the best configuration: Mean Average Precision gives an overall performance value, but it could be not so significative from the point of view of the user. For example, a client could ask to find the 10 or the 50 Nearest Neighbors of a query image, usually no more: even a system that has a very poor *mAP* value but contains a lot of examples per each class could give back to the user only relevant results.

To this aim, we compute P@k and R@k for our best configuration, considering some typical k values. Those values have been calculated for each query (one query per class) and averaged, leading to the following formulas:

$$mP@k = \frac{1}{Ck}\sum_{i=1}^{C} Relevant\ Retrieved\ Documents_i$$

$$mR@k = \frac{1}{Ck}\sum_{i=1}^{C} Retrieved\ Documents_i \cap Groundtruth\ NN_i$$

where C is the number of database classes. As ground truth NN we considered the ones returned by the brute force approach. On Figure 11 we report the values for different amounts of k.

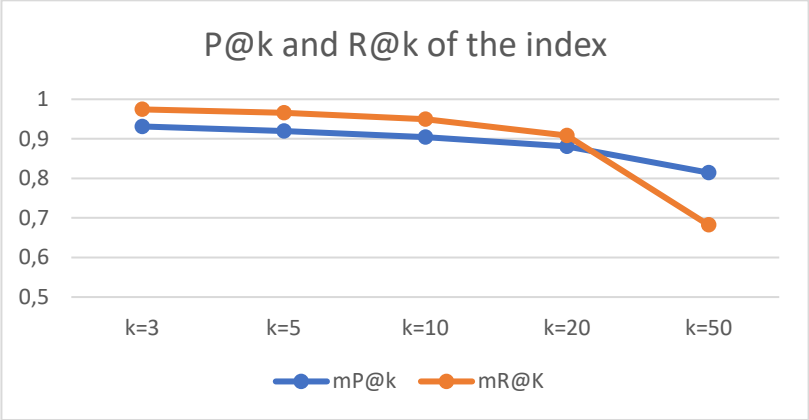


Figure 11. P@k and R@k of the index

As expected, performance tends to degrade when we increase k, but they are in general high.

For $k = 3$, both P@3 and R@3 are very close to 1, meaning that there is a very low probability respectively to find a non-relevant result among the retrieved ones (0.068) and to find a result that should not be ranked between the first 3 nearest neighbors (0.022). The value of R@r is initially awesome, but it degrades quickly, meaning that for big k values, the model struggles in keeping the ranking consistent with the one of the ground truth. The fact that P@r remains always above 0.8 highlights how this configuration succeeds in retrieving relevant results for the user: even if the *Mean Average Precision* is 0.677, in average more than the 80% of the results given back to the user at each query will be relevant.

5. ERROR ANALYSIS

By evaluating the retrieval performance of our image search engine, we were able to conclude that, for most species, the performance was good or at least acceptable. However, there are some species for which the *AP* was very poor (below 0.1) both using the index and the brute force approach. We believe that this poor performance is mainly due to the query image that was selected for that species. In fact, by choosing the first image from the test set for each species, it may have happened that the query image contained the bird in an unusual pose or that it did not highlight the distinctive features of that species. This has brought us to carry out an analysis of the errors committed by our system, in order to understand which are its weak points, that is with which queries it does not perform well.

5.1. Bird sex

As we had already anticipated, most of the images within the dataset (and consequently the training set) represent male examples. This imbalance led the deep neural network to learn to recognize characteristics of the male gender of the species more accurately, rather than those of the female gender. In turn, this causes features extracted from male bird images to be more meaningful than those extracted from female bird images and affects the retrieval performance of the search engine. Therefore, in most cases where we present the search engine with an image of a female bird, the performance is not optimal.

An example of this can be seen in Figure 12, in which the results obtained for a female bobolink as query are depicted. In this species, while males are mostly black with creamy napes and white scapulars, lower backs, and rumps, females are mostly light brown, with black streaks on the back and flanks, and dark stripes on the head.

The image search engine can correctly retrieve images containing female bobolinks, especially in the first few results. However, right from the start and then increasingly frequently it also retrieves images of birds belonging to other species. Out of 100 recovered images, only 21 are relevant, resulting in a $precision@100(female) = 0.21$.

Regardless, the system retrieves images of birds with similar characteristics and, more interestingly, many of the retrieved images represent female examples (as can be clearly seen with the lark buntings).

Figure 12. AP and (some) of the results obtained by presenting an image of a female bobolink as query

The *Average Precision* in this case is very low, 0.0998, but the same does not happen when we present the search engine with an image of a male bobolink, as can be seen in Figure 13. In this case, the system works well achieving an

Average Precision of 0.6032. Moreover, out of 100 images, 88 are relevant, resulting in a $precision@100(male) = 0.88$.

Figure 13. AP and (some) of the results obtained by presenting an image of a male bobolink as a query

5.2. Bird pose

Another thing we have noticed impacting the performance of the search engine is the pose in which the bird in the query image is portrayed. As a matter of fact, while it is true that for many species there are several images of birds flying, there are some species for which most of the images depict the bird standing still, sitting on a branch or on the ground. For these particular species, the search engine does not perform well if a query image containing a bird in flight is presented, and often it retrieves images of birds in the air, regardless of the species.

As an example, we can see the results shown in Figure 14 obtained after presenting an image of a flying lilac roller as a query. In this case, the retrieval performance is very poor, with an *Average Precision* of 0.0271. This is because, as we can see, the system focuses more on the fact that the bird is flying rather than on the distinctive features of the species, such as plumage colors. The number of retrieved images that are relevant is really low, resulting in a $precision@100(flying) = 0.12$.

By all means, it is interesting to note that the lilac roller is often mistaken by the system for an Indian roller, another species belonging to the same family, Coraciidae.

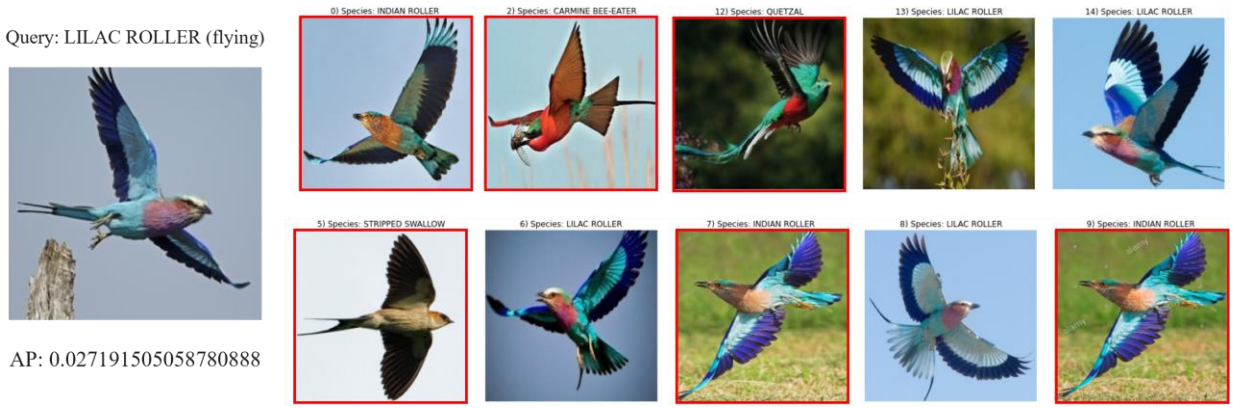


Figure 14. AP and (some) of the results obtained by presenting an image of a flying lilac roller as a query

By changing the query image and using one where the bird is sitting, we achieve much better results. As a matter of fact, as we can see from Figure 15, the search engine achieves an *Average Precision* of 0.8251. In addition, out of the 100 images retrieved, 85 are relevant, resulting in a $precision@100(sitting) = 0.85$.

Figure 15. AP and (some) of the results obtained by presenting an image of a sitting lilac roller as a query

6. EXPLAINABILITY

We will now devote our attention to understanding how our networks classify images, to try to understand the root of the most common errors and to understand why the average precision of some classes outperforms other classes. We decided to use the heatmap of class activation technique. This technique is useful to understand which parts of a given image led to a convolutional network to its final classification decision. A “class activation” heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class considered.

First, we want to try to understand why feature extraction of ResNet-50v2 alone works so poorly. In fact, for now we have given some personal considerations, but we want to understand why the network suffers in the extraction of good features. To do this, we considered the classes that perform the worst in terms of mean average precision, and calculated the heatmaps, overimposing them on the original image. For each class we considered the same image that had been used as a query in previous chapters.

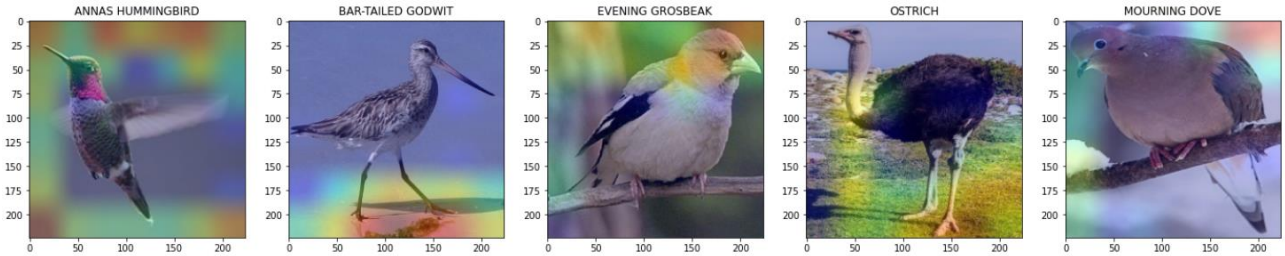


Figure 16. Heatmaps for the worst classified classes of feature extraction

As we can easily see, the network focuses heavily on background information, such as the sky and the terrain. This is most noticeable in the first two images, in which the network does not seem to consider the birds at all.

We will now analyze how ResNet152 behaves with the addition of the classification MLP. We will report in this case both the best and the worst results, always in the form of heatmaps.

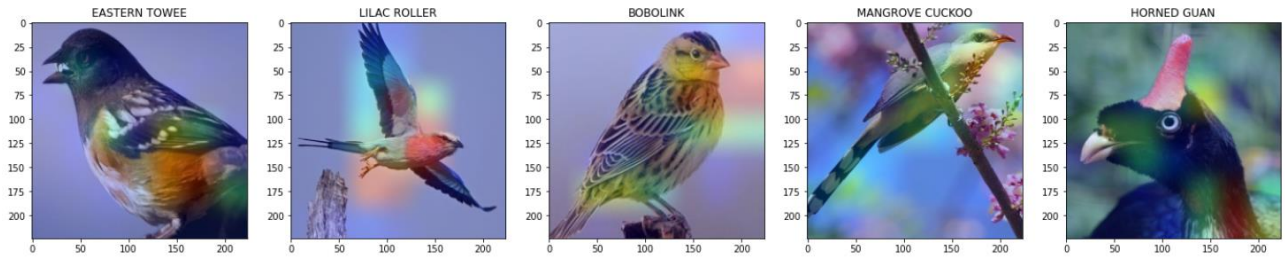


Figure 17. Heatmaps for the worst classified classes (MLP on top of ResNet-152v2)

As we can see in Figure 17, in some classes again the network focuses a little too much on background details, this can be easily seen in the case of the bobolink and the lilac roller. These two birds have already been analyzed in the chapter on error analysis, and the main problem is the fact of being a female for the first and the pose in flight for the second.

In the heatmaps of the best classes (Figure 18), we can see how well the network focuses on details internal to the birds and manages to distinguish distinct properties for each of them. For example, it is interesting to note how it focuses on the peacock's eyespots, or how it focuses on the pelican's beak.

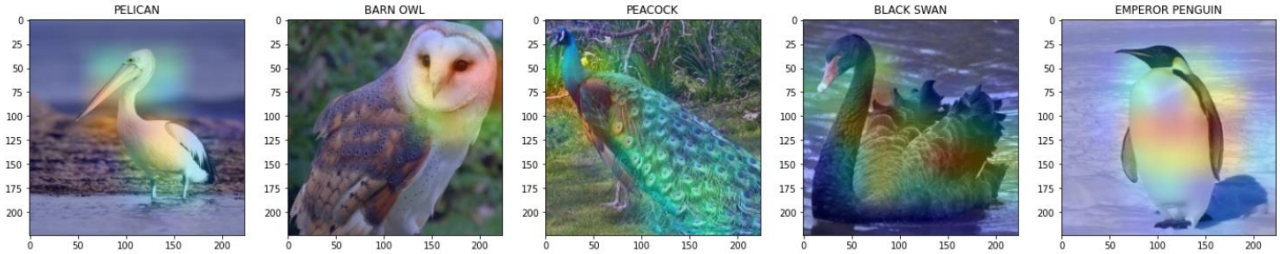


Figure 18. Heatmaps for the best classified classes (MLP on top of ResNet-152v2)

Finally, we also report in Figure 19 and Figure 20 the results after fine tuning the convolutional networks. With fine tuning we can see more precision in the details, although the errors remain almost the same. In the case of the maleo it actually focuses on internal details but confuses it with other similar birds.

As for the best results, we can see some incredible outcomes, such as in the case of the african crowned crane, whose crown – which is the most distinctive feature of this bird – is analyzed.

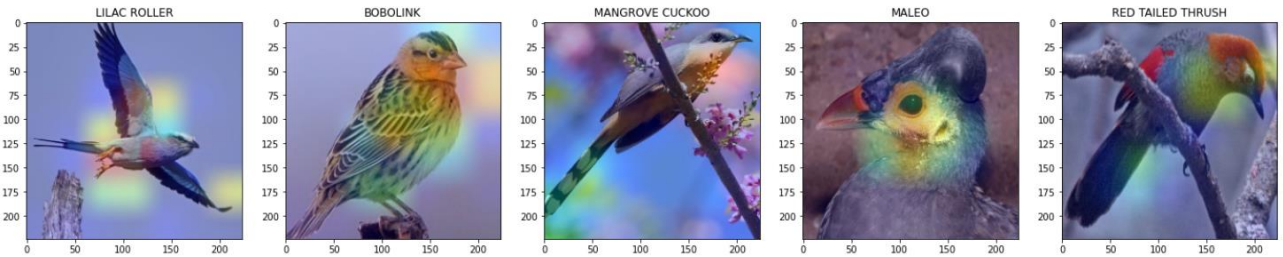


Figure 19. Heatmaps for the worst classified classes (Fine tuning)

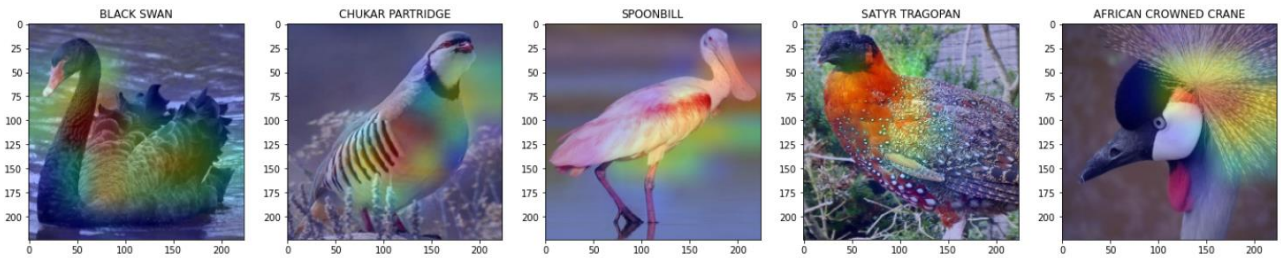


Figure 20. Heatmaps for the best classified classes (Fine tuning)

7. FINAL IMAGE SEARCH ENGINE AND WEB INTERFACE

Although the best configuration of ResNet model, distance/similarity and PP-Index improvement(s) was found in previous chapters, we decided to give the user the possibility to use different configurations:

- *Model*: the user can choose to use either the feature extraction or the fine-tuning model. Regarding the feature extraction, the model used is obtained from the combination of the model ResNet152v2 with a dense layer of size 512, and the encoding part of the autoencoder that reduces the features to 128, since we have seen that this operation is beneficial. Concerning fine-tuning, the network used is also a combination of ResNet152v2 with a 512-dimension dense layer (in this case, fine-tuned as before described) and the encoding part of the autoencoder that reduces the features to 128.
- *Metric*: the user can choose between Euclidian distance or cosine similarity.
- *PP-Index improvements:* the user can decide to use one or both of the enhancements we implemented for the PP-Index (query perturbation and forest)
- *Number of images retrieved:* the user can change the number of images to receive in response to a query by modifying the value of the k parameter

The performances concerning the different configurations are summarized in Figure 21 regarding the feature extraction model and in Figure 22 for fine tuning. Please note that $k = 250$ was used for these calculations as before.

As can be seen from the two graphs, indexes built on features extracted from the finetuned model always perform better than those built on features extracted from the non-finetuned one. Also, the best metric is always cosine similarity.

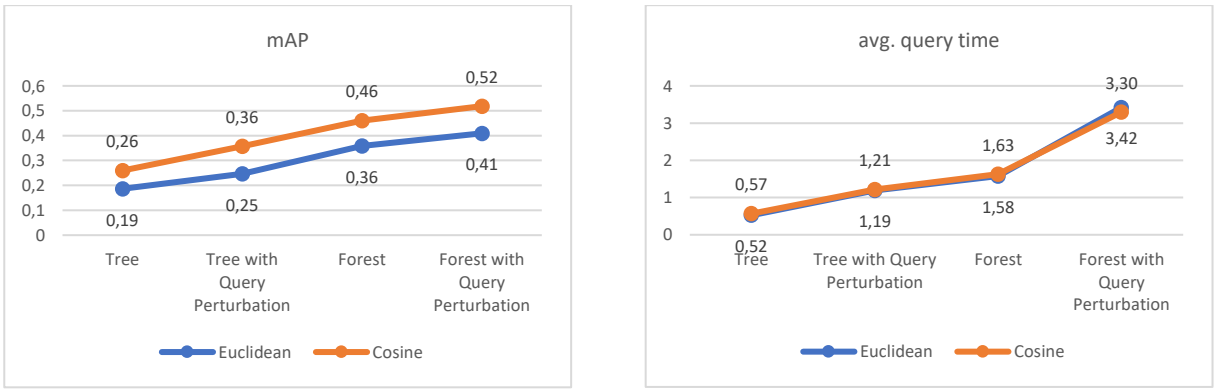


Figure 21. Mean Average Precision and Average Query Time for various index implementations (Feature Extraction)

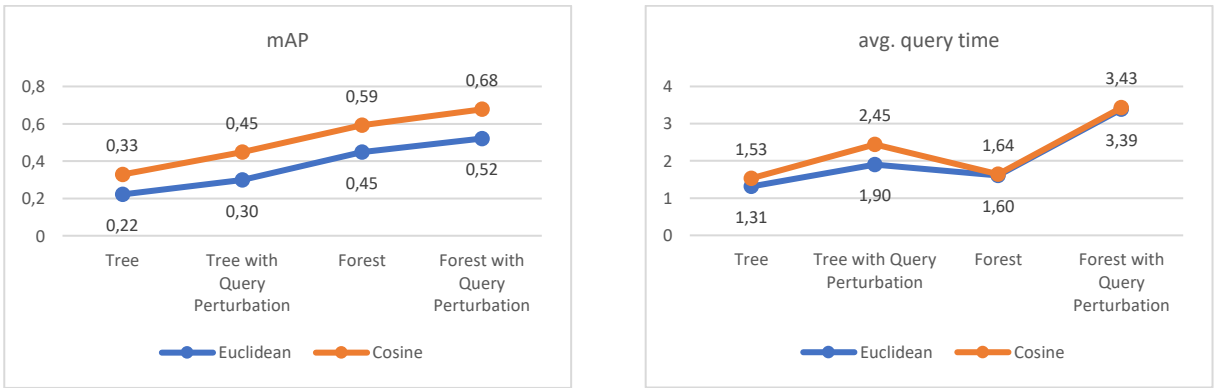


Figure 22. Mean Average Precision and Average Query Time for various index implementations (Fine Tuning)

As for the development of the web interface, we decided to use Anvil[5], a framework for building full-stack web applications with nothing but Python. The web page is hosted directly by Anvil and the server code, which allows the search engine to work, is written on Colab.

Our image search engine is publicly available at https://birds-search-engine.anvil.app/ and Figure 23 shows it in action. As can be seen, and as we had anticipated, the user can select the different configurations of model, k, metric, and optimization(s). Furthermore, the user can choose whether to have the distances of each image from the query shown or not. The web interface also prints a summary of the configuration choices below the "*Load your image here*" label.

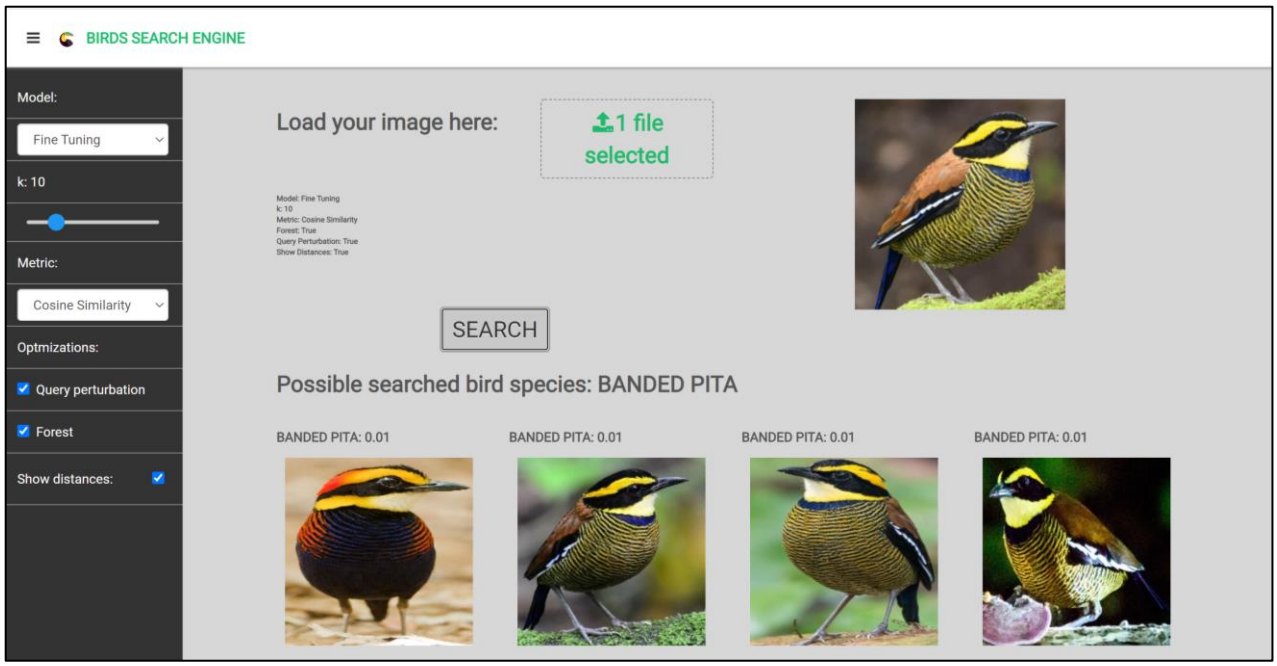


Figure 23. Web interface

Once users submit their image to the system, they are returned not only the first k images that are most similar to the query, but also a possible prediction of the species to which the query bird belongs. This classification is done through a *weighted k-nearest-neighbors*. Specifically, the score of membership of the object o to the species s is calculated as:

$$score(s,o) = \sum_{o' \in N_k(o)} I_c(o') \cdot \cos(o', o), \qquad \text{if cosine similarity has been chosen}$$

$$score(s,o) = \sum_{o' \in N_k(o)} I_c(o') \cdot \frac{1}{euclidean(o', o)}, \qquad \text{if Euclidean distance has been chosen}$$

where $N_k(o)$ is the set of k nearest neighbors of o and $I_c(o') = 1$ if o belongs to species s and 0 otherwise. The query image is then assigned to the species with the highest score.

We chose to employ this technique because typically weighting by similarities is more accurate than simple voting that takes into account only cardinality. Moreover, we already know the distance (or similarity) of each of the neighbors with respect to the query and therefore this does not require additional costs.

[5] https://anvil.works/